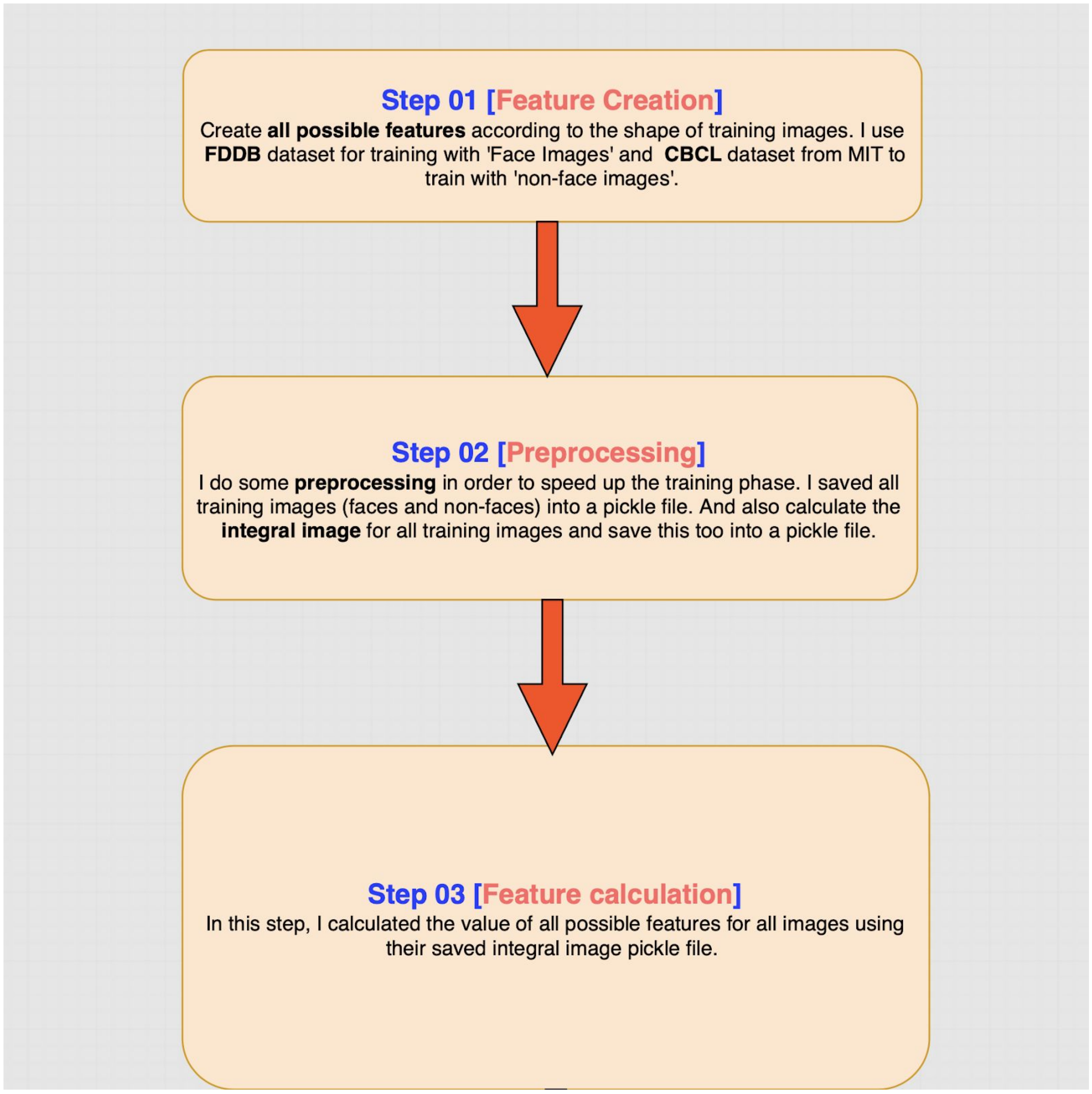
CSE 573 : UB ID : **50291708**

Face Detection in the Wild

Detecting frontal faces in image using Viola_Jones face detection algorithm

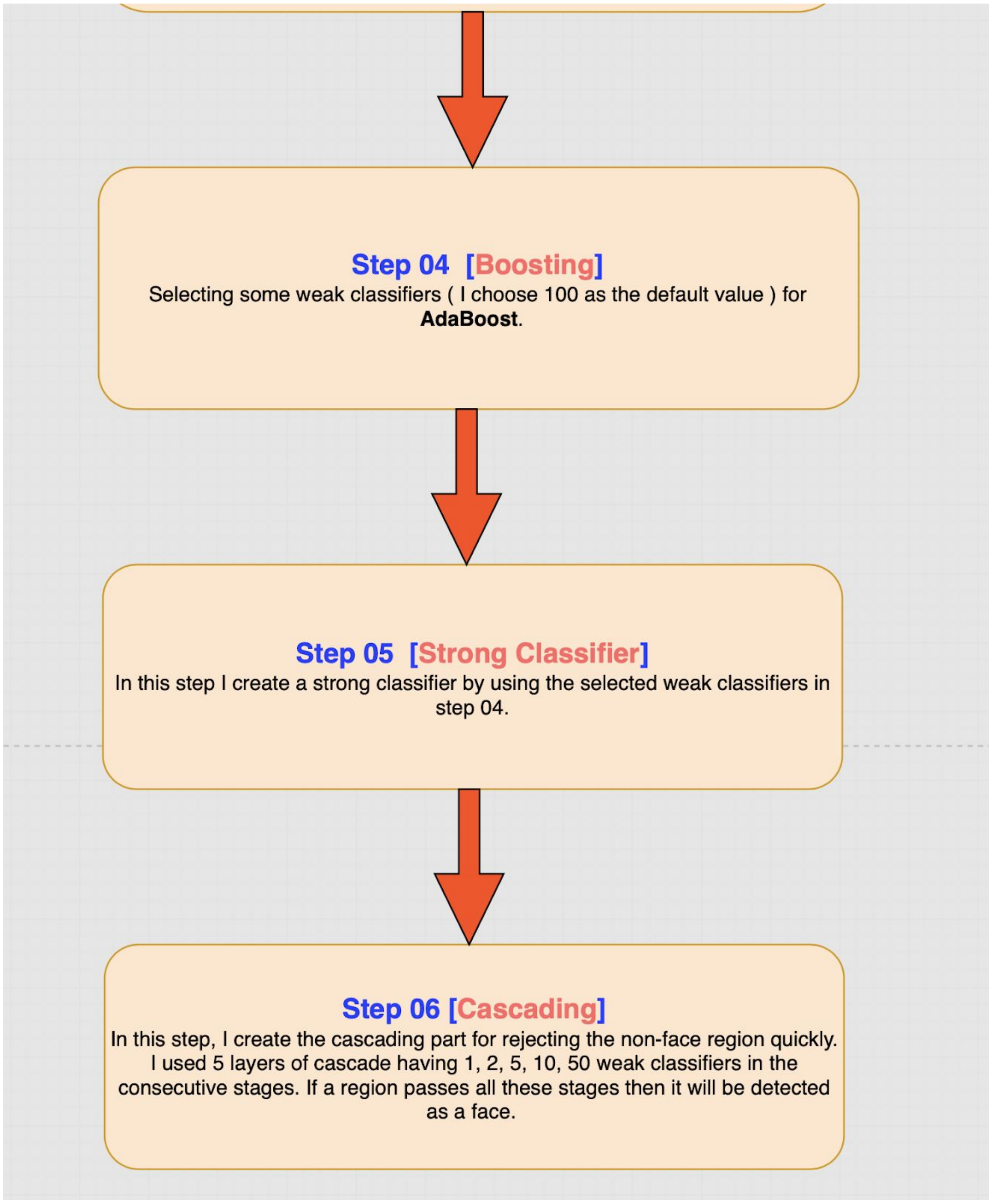
Flow chart of my workflow :



Step 01 [Feature Creation]
Create **all possible features** according to the shape of training images. I use **FDDB** dataset for training with 'Face Images' and **CBCL** dataset from MIT to train with 'non-face images'.

Step 02 [Preprocessing]
I do some **preprocessing** in order to speed up the training phase. I saved all training images (faces and non-faces) into a pickle file. And also calculate the **integral image** for all training images and save this too into a pickle file.

Step 03 [Feature calculation]
In this step, I calculated the value of all possible features for all images using their saved integral image pickle file.

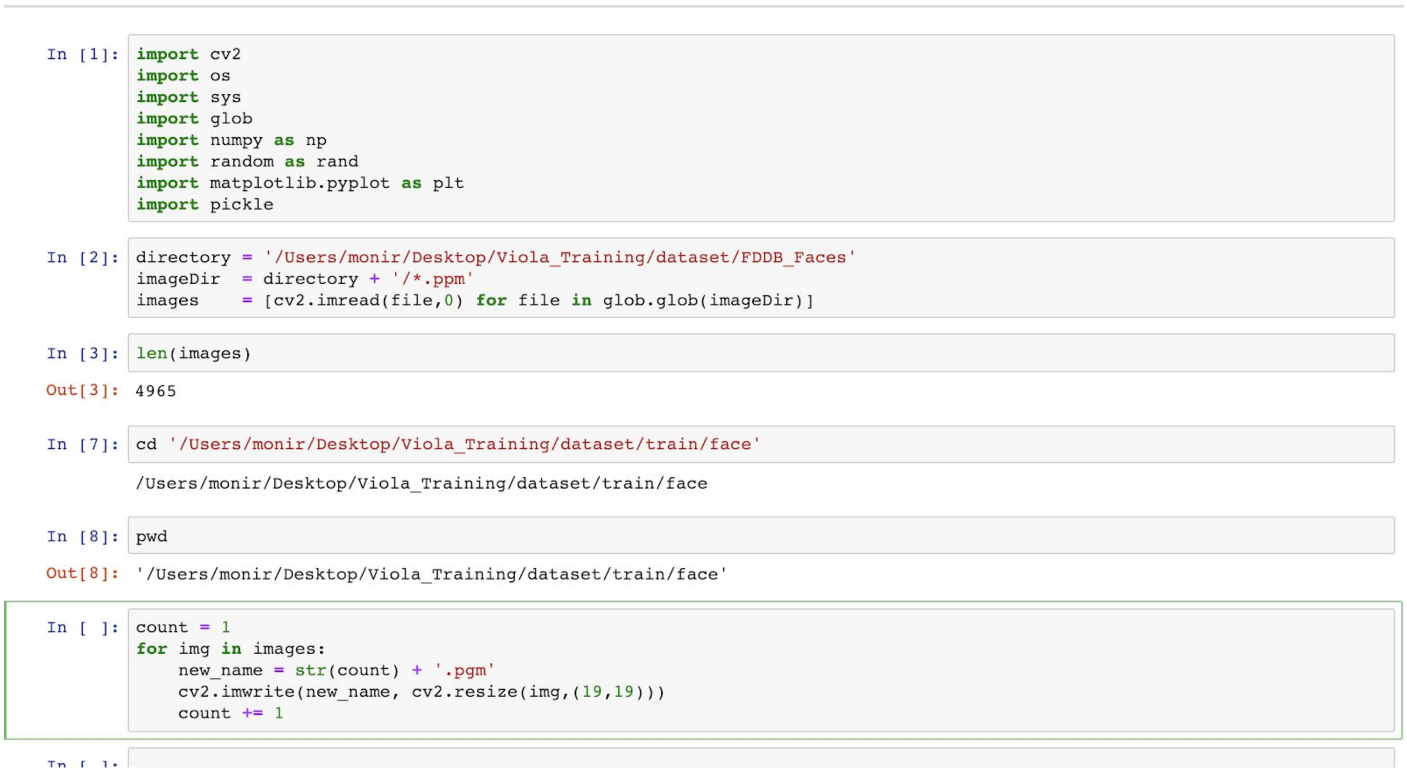


Step 04 [Boosting]
Selecting some weak classifiers (I choose 100 as the default value) for **AdaBoost**.
Step 05 [Strong Classifier]
In this step I create a strong classifier by using the selected weak classifiers in step 04.
Step 06 [Cascading]
In this step, I create the cascading part for rejecting the non-face region quickly. I used 5 layers of cascade having 1, 2, 5, 10, 50 weak classifiers in the consecutive stages. If a region passes all these stages then it will be detected as a face.

Step 01 :

I combined the ‘face’ images from 2 datasets : FDDB and CBCL. As there was no ‘non-face’ images in FDDB dataset that’s why I selected the CBCL dataset as it has ‘non-face’ images for training. I used a total of **7394 face** and **4584 non-face** images. And I generated almost 52k features by using a window size of 19 by 19. I need to resize all the images from FDDB dataset as there image size was 86 by 86. I resized them all to 19 by 19 so that can merge the two datasets. [My Training Dataset : Link]

Modification code for FDDB :



```
In [1]: import cv2
        import os
        import sys
        import glob
        import numpy as np
        import random as rand
        import matplotlib.pyplot as plt
        import pickle

In [2]: directory = '/Users/monir/Desktop/Viola_Training/dataset/FDDB_Faces'
        imageDir  = directory + '/*.ppm'
        images    = [cv2.imread(file,0) for file in glob.glob(imageDir)]

In [3]: len(images)

Out[3]: 4965

In [7]: cd '/Users/monir/Desktop/Viola_Training/dataset/train/face'

        /Users/monir/Desktop/Viola_Training/dataset/train/face

In [8]: pwd

Out[8]: '/Users/monir/Desktop/Viola_Training/dataset/train/face'

In [ ]: count = 1
        for img in images:
            new_name = str(count) + '.pgm'
            cv2.imwrite(new_name, cv2.resize(img,(19,19)))
            count += 1
```

I created **5 types of features** for all possible scales and positions. All this codes are in feature.py file in my submission.

Rectangle Class - A helper class for `Haar Feature` calculation.

A Haar Feature is a collection of Rectangle Regions.

7]:
```
class Rectangle:
    # constructor
    def __init__(self, x, y, width, height):
        self.x = x
        self.y = y
        self.width = width
        self.height = height

    # Return the sum of all pixels inside a rectangle for a specific integral image
    def compute_sum(self, integralImg):

        x = self.x
        y = self.y
        width  = self.width
        height = self.height

        one   = integralImg[y][x]
        two   = integralImg[y][x+width]
        three = integralImg[y+height][x]
        four  = integralImg[y+height][x+width]

        desiredSum = (one + four) - (two + three)

        return desiredSum
```

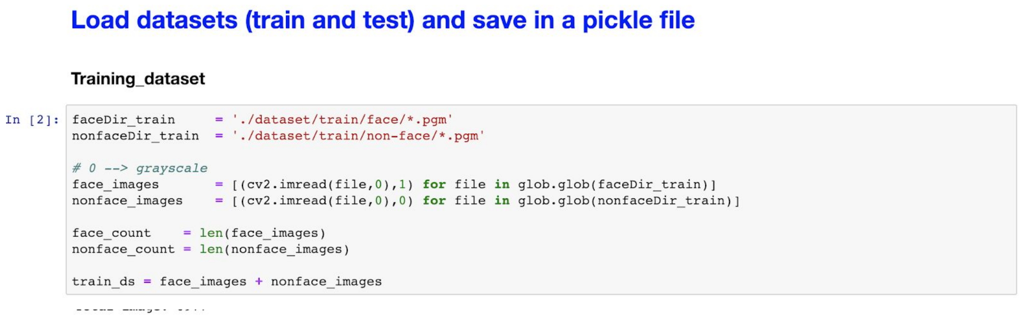
Creating all possible features in `19 by 19` window size

Feature Class - A helper class for creating `Haar Feature`

:n [8]:
```
class Feature:
    # constructor
    def __init__(self, image_shape):

        self.height, self.width = image_shape
        self.f = None  # Feature list
        self.f_values = None # Features' values for all images

    def creating_all_features(self):
        '''
          Create 5 types of Haar Features for all sizes, shapes and positions in a fixed window
        '''
        height = self.height
        width  = self.width

        # List of tuple where 1st element means List of black rectangles and 2nd element means List of white rectangles
        features = []

        for w in range(1, width+1):      # All possible width
            for h in range(1, height+1): # All possible height

                i = 0
                while i + w < width:
                    j = 0
                    while j + h < height:

                        fixed   = Rectangle(i, j, w, h)
                        right_1 = Rectangle(i+1*w, j, w, h)
                        right_2 = Rectangle(i+2*w, j, w, h)

                        bottom_1_right_1 = Rectangle(i+1*w, j+1*h, w, h)

                        bottom_1 = Rectangle(i, j+1*h, w, h)
                        bottom_2 = Rectangle(i, j+2*h, w, h)

                        '''
                          2 Rectangle Haar Features
                        '''
                        # Horizontal  -->  fixed (white) | right_1 (black)
                        if i + 2 * w < width:
                            features.append(([right_1], [fixed]))

                        # Vertical -->  fixed(black)
                                    #  ------------
                                    #   bottom_1(white)
                        if j + 2 * h < height:
                            features.append(([fixed], [bottom_1]))


                        '''
                          3 Rectangle Haar Features
                        '''
                        # Horizontal -->  fixed (white) | right_1 (black) | right_2 (white)
                        if i + 3 * w < width:
                            features.append(([right_1], [right_2, fixed]))

                        # Vertical -->  fixed(white)
```

Step 02 :

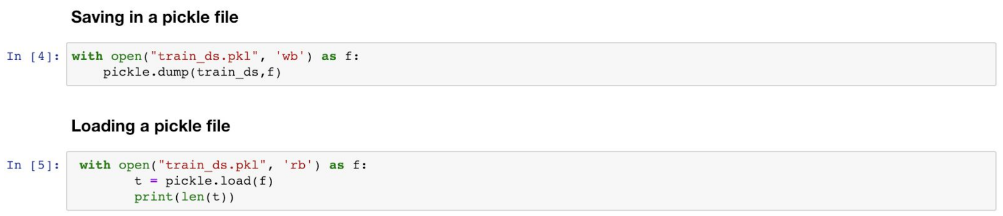
As the training part is very slow for viola-jones model, so I tried to speed up the process by doing some preprocessing. I saved all training images (faces and non-faces) into a pickle file. And also calculate the integral image for all training images and save this too into a pickle file. So this part is not tied to my training code. I just create this once and it makes the whole training part a little easier.



Load datasets (train and test) and save in a pickle file

Training_dataset

```
In [2]: faceDir_train     = './dataset/train/face/*.pgm'
        nonfaceDir_train  = './dataset/train/non-face/*.pgm'

        # 0 --> grayscale
        face_images       = [(cv2.imread(file,0),1) for file in glob.glob(faceDir_train)]
        nonface_images    = [(cv2.imread(file,0),0) for file in glob.glob(nonfaceDir_train)]

        face_count    = len(face_images)
        nonface_count = len(nonface_images)

        train_ds = face_images + nonface_images
```



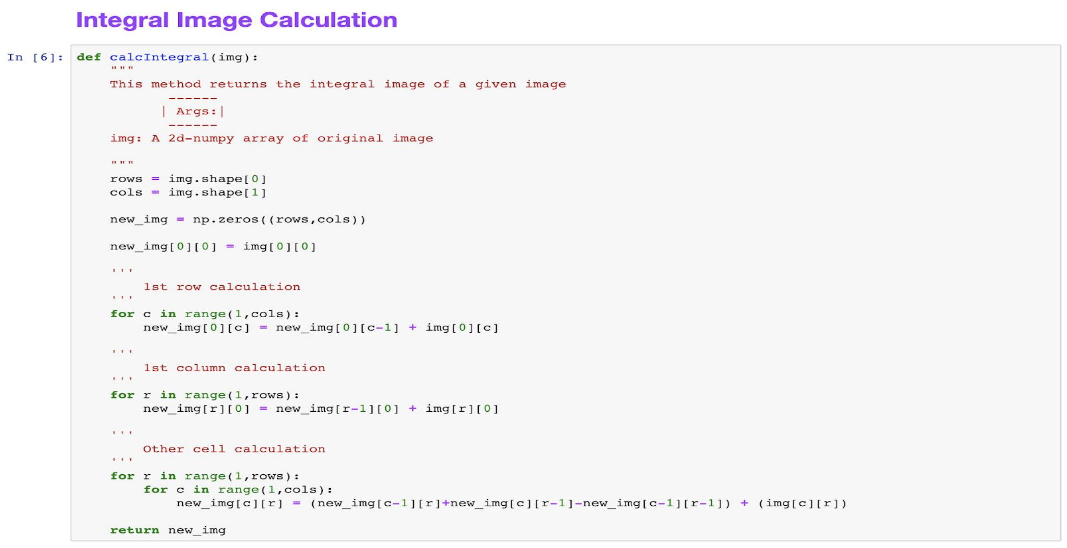
Saving in a pickle file

```
In [4]: with open("train_ds.pkl", 'wb') as f:
            pickle.dump(train_ds,f)
```

Loading a pickle file

```
In [5]:  with open("train_ds.pkl", 'rb') as f:
                t = pickle.load(f)
                print(len(t))
```

Integral Image Calculation :

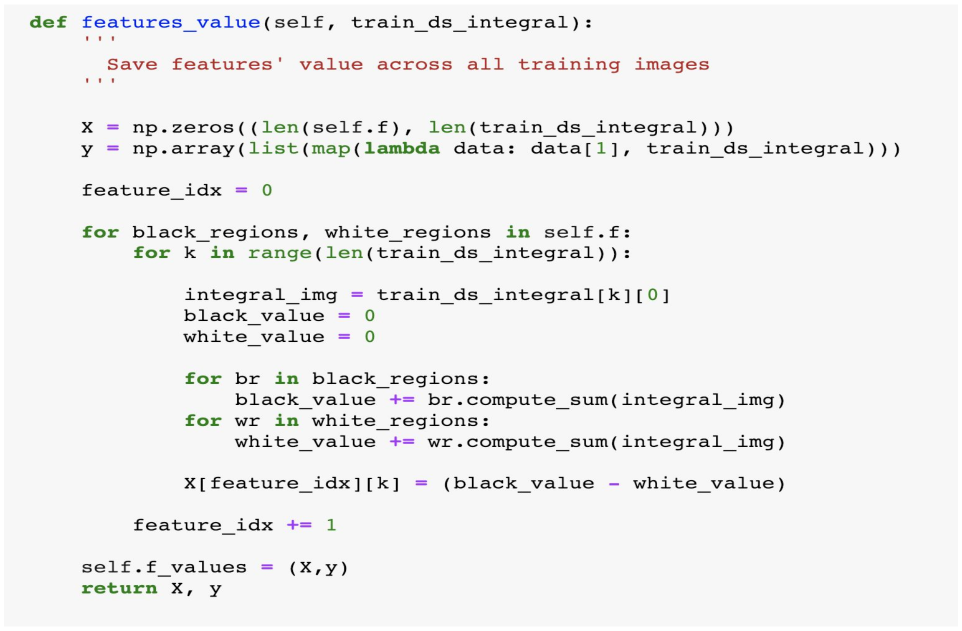


Integral Image Calculation

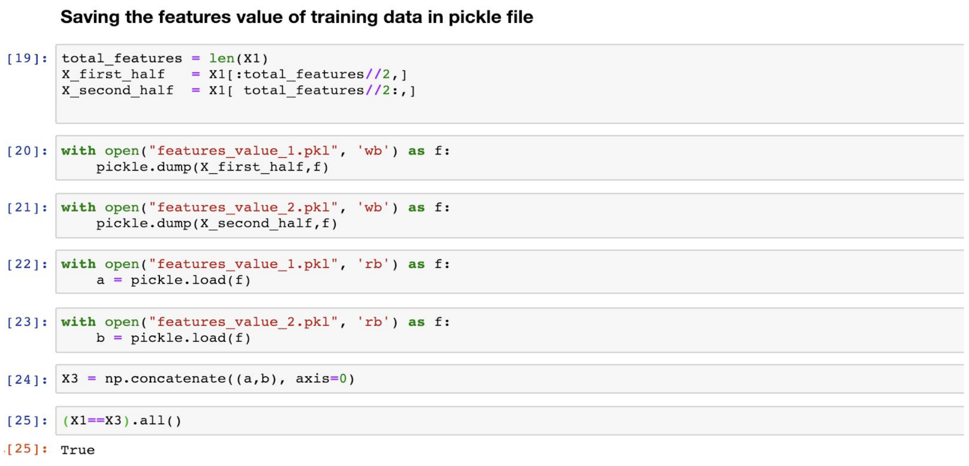
```
In [6]: def calcIntegral(img):
            """
            This method returns the integral image of a given image
                   ------
                  | Args:|
                   ------
            img: A 2d-numpy array of original image

            """
            rows = img.shape[0]
            cols = img.shape[1]

            new_img = np.zeros((rows,cols))

            new_img[0][0] = img[0][0]

            '''
                1st row calculation
            '''
            for c in range(1,cols):
                new_img[0][c] = new_img[0][c-1] + img[0][c]

            '''
                1st column calculation
            '''
            for r in range(1,rows):
                new_img[r][0] = new_img[r-1][0] + img[r][0]

            '''
                Other cell calculation
            '''
            for r in range(1,rows):
                for c in range(1,cols):
                    new_img[c][r] = (new_img[c-1][r]+new_img[c][r-1]-new_img[c-1][r-1]) + (img[c][r])

            return new_img
```

Step 03 :

I used a total of 7394 face and 4584 non-face images. And I generated almost 52k features by using a window size of 19 by 19. So, I have a matrix of size **11978 by 52000.** Here, each cell denotes the value of a particular feature for a particular image. This step is also time-consuming. For speeding up the whole training phase, I saved this matrix into a pickle file. But this was more than 3 GB so I divided the matrix into two halves and save each half into a pickle file. (ex. features_value_1.pkl, features_value_2.pkl).



```
def features_value(self, train_ds_integral):
    '''
      Save features' value across all training images
    '''

    X = np.zeros((len(self.f), len(train_ds_integral)))
    y = np.array(list(map(lambda data: data[1], train_ds_integral)))

    feature_idx = 0

    for black_regions, white_regions in self.f:
        for k in range(len(train_ds_integral)):

            integral_img = train_ds_integral[k][0]
            black_value = 0
            white_value = 0

            for br in black_regions:
                black_value += br.compute_sum(integral_img)
            for wr in white_regions:
                white_value += wr.compute_sum(integral_img)

            X[feature_idx][k] = (black_value - white_value)

        feature_idx += 1

    self.f_values = (X,y)
    return X, y
```



Saving the features value of training data in pickle file

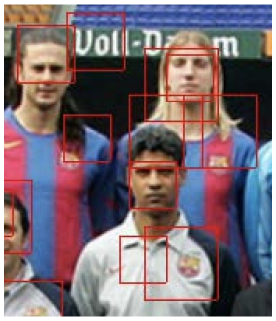
```
[19]: total_features = len(X1)
      X_first_half   = X1[:total_features//2,]
      X_second_half  = X1[ total_features//2:,]

[20]: with open("features_value_1.pkl", 'wb') as f:
          pickle.dump(X_first_half,f)

[21]: with open("features_value_2.pkl", 'wb') as f:
          pickle.dump(X_second_half,f)

[22]: with open("features_value_1.pkl", 'rb') as f:
          a = pickle.load(f)

[23]: with open("features_value_2.pkl", 'rb') as f:
          b = pickle.load(f)

[24]: X3 = np.concatenate((a,b), axis=0)

[25]: (X1==X3).all()

[25]: True
```

Step 04 :

In this step, I mainly implement weak classifier training and best weak classifier selection. Suppose I chose to use 5 weak classifiers. I need to find out the best 5 weak classifiers. I need 5 round for that purpose. I initialize the weights of the training images. Then in every round, I train all weak classifiers that means find threshold and polarity for every weak classifier. Then select the best weak classifier.

Step 05 & 06:

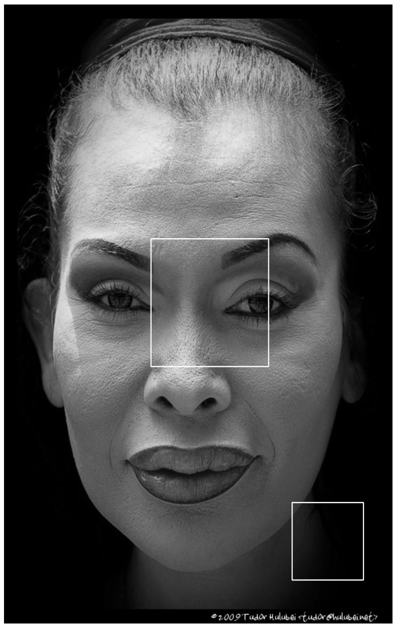
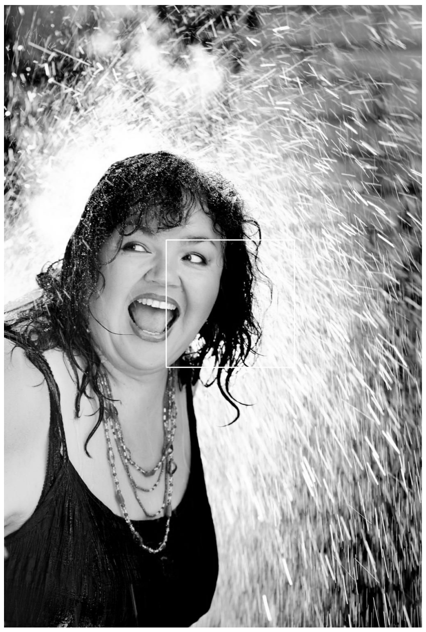
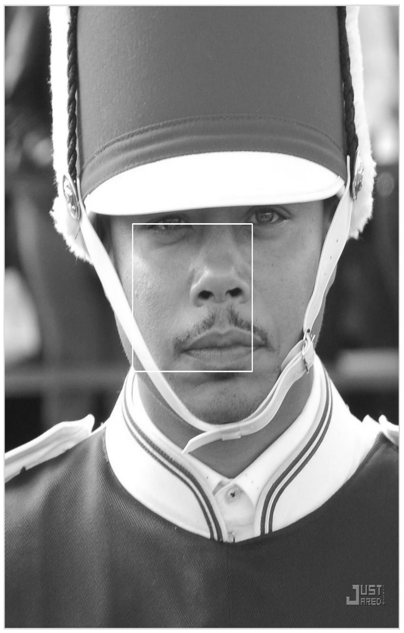
I follow the pseudocode from the paper exactly and implement those pseudocodes. In step 05, I mainly implement weak classifier training and best weak classifier selection. Suppose I chose to use 5 weak classifiers. I need to find out the best 5 weak classifiers. I need 5 round for that purpose. I initialize the weights of the training images. Then in every round, I train all weak classifiers that means find threshold and polarity for every weak classifier. Then select the best weak classifier. In step 06 I created a cascade of classifiers to detect a non-face region quickly. I used the technique “Non-maximal Suppression” to remove the overlapping windows like the image below.

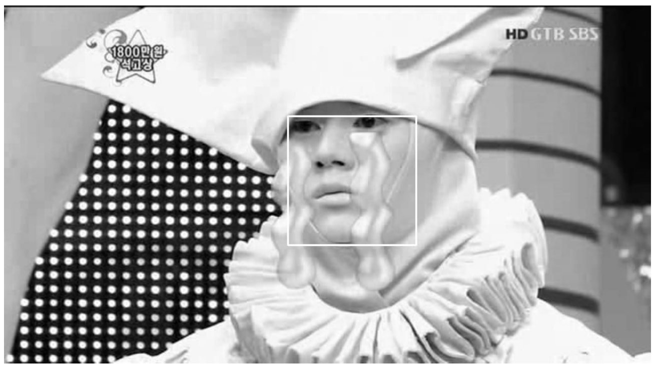


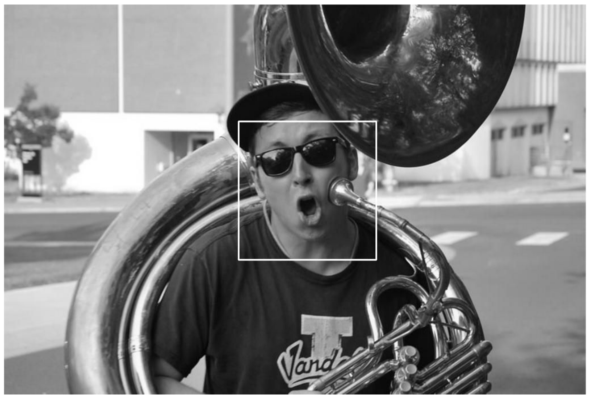
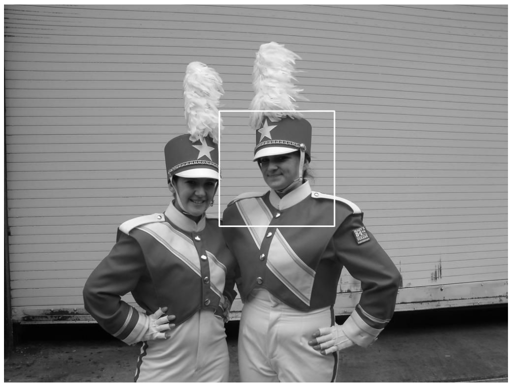
Some of My Output for the given 1000 test images :

I draw the rectangles for every test images. Here are some of the images :

Output of my model for all 1000 test images : Link



Analysis of my Results :

I figure the failure cases of my model as I checked the output for all 1000 images. I trained my model for 19 by 19 window size. So, when I try to detect a face in a large image I need to rescale my size so that it can detect face of any sizes. **I don't resize the image rather I rescaled my window and accordingly all features for that window**. I used a scaling factor of **1.5**. But I realize that I can't scale the features properly when I am re-scaling the window size. That's why there are some obvious false positives in my outputs. The bounding box is not scaled properly according to the window scaling. This is one issue I figured. The possible improvement is to figure out how to effectively re-scale all the features according to the scaling of window.

Another error I figured is that there are many faces which are not strictly frontal face. As we trained with only frontal faces so my model can't detect those and as a result gives a lot of false positives in those images.

Also, I use **7394 face** and **4584 non-face** images for training. There are not enough non-face images for the training compared to face images. That's one of the reason of so many false positives. If I can use more non-face images for training then the amount of false positive will reduce a lot.

Result of my face detector on FDDB :

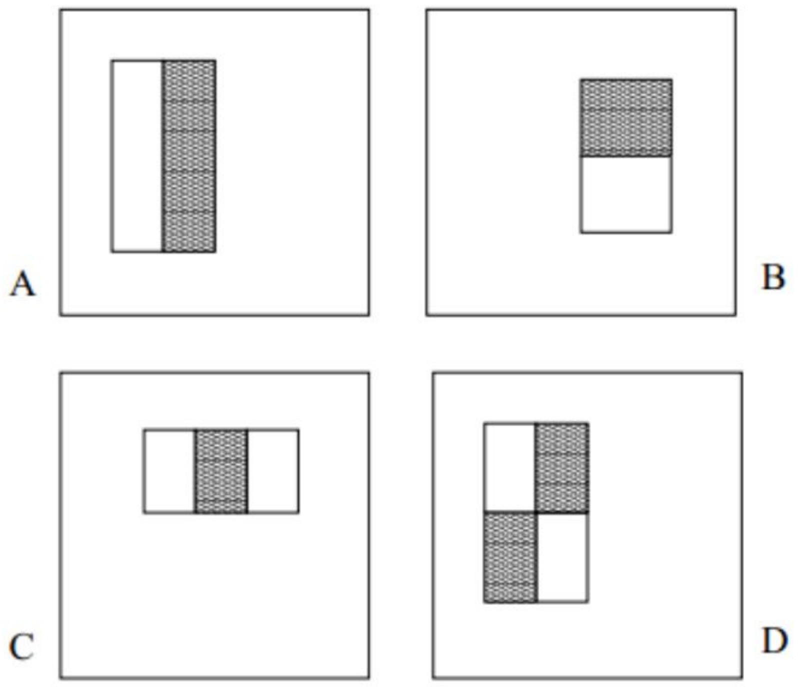
Initially when I was using FDDB dataset and CBCL dataset (non-faces only) for training my model, I didn't use almost 900 images from FDDB for training. I later used those images for testing my model. I got almost 92% accuracy for those 900 images which I didn't use for training my model. But later when I used my model to detect faces in the given 1000 test images my model was performing very poor as there were lots of false positives. Then I used all FDDB images and CBCL dataset (faces and non-faces all images) to train my model again. After that the performance has improved a lot. It took almost **9 hour** to train the model for all the training images.

Overview of VIOLA_JONES algorithm :

Based on Robust Real-time Object Detection Framework presented by Paul Viola & Michael Jones, there are 3 steps to achieve Face Detection through a frame:

1. Integral Image representation which speeds up computation of features.
2. A small number of significant features using AdaBoost build the classifier
3. A cascade structure combined by several more complex classifiers make the detector focus on the **promising regions** so that an object could be located in much shorter time.

Feature Computation :



Three types of rectangle features. The sum of the pixels which lie in the white rectangles are subtracted from the sum of pixels in the grey rectangles. (A) and (B) show **two rectangle features.** (C) shows a **three-rectangle** feature. (D) shows a **four-rectangle** feature

The value of the integral image at point (x,y) is the sum of all the pixels above and to the left.

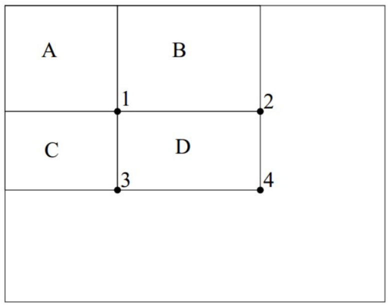
$$ii(x,y) = \sum_{x' \leq x, y' \leq y} i(x', y'),$$

Where *ii(x,y)* is the integral image and *i(x,y)* is the original image

$$s(x,y) = s(x,y-1) + i(x,y)$$

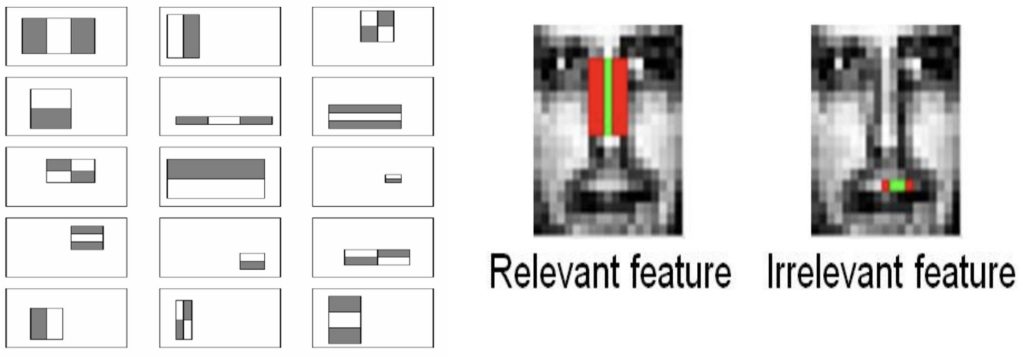
$$ii(x,y)=ii(x-1,y)+s(x,y)$$

where s(x,y) is the cumulative row sum



The sum of the pixels within rectangle D can be computed with four array references. For example, ii(x2,y2) = sum(A) + sum(B),in this case, Sum(D)=ii(x4,y4)+ii(x1,y1)-ii(x2,y2)- ii(x3,y3).

<u>Feature Selection :</u>

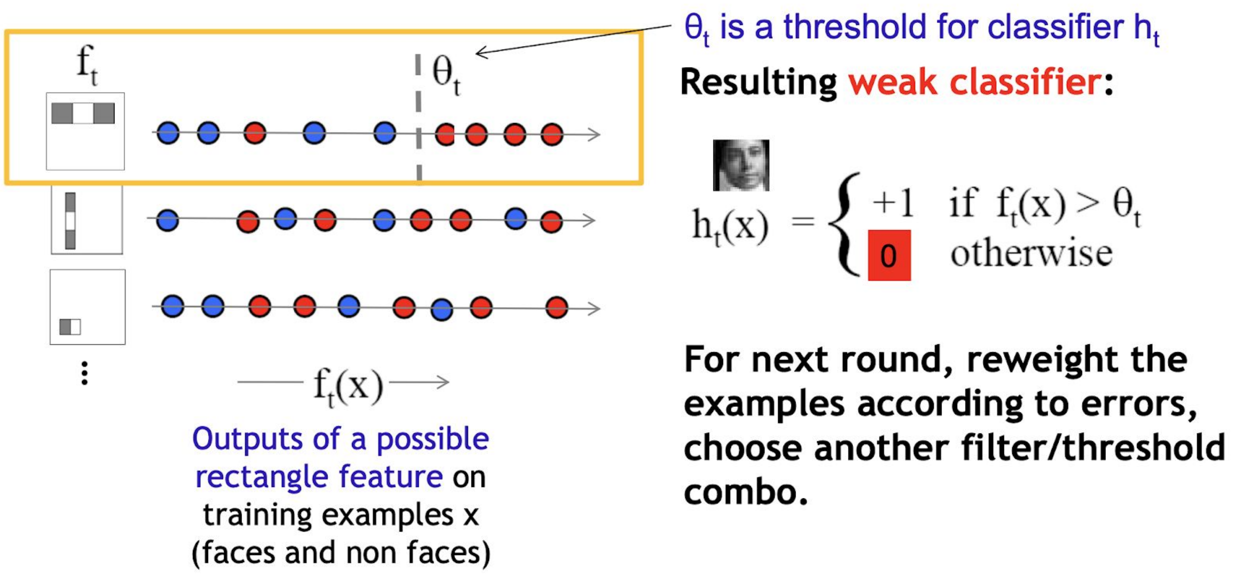


To avoid expensive computation spent on all rectangle features of each sub-window, a very small subset of features would be selected to form an effective classifier. That is, in Viola–Jones object detection framework, relevant features will be chosen and classifier built by Adaboost. AdaBoost learning algorithm makes a linear combination of “weak” classifiers to realize a “strong” classifier.

$$\underset{\text{Strong classifier}}{F}(\underset{\text{Image}}{x}) = \underset{\text{Weight}}{\alpha_1}\underset{\text{Weak classifier}}{f_1}(x) + \alpha_2 f_2(x) + \alpha_3 f_3(x) + \dots$$

Adaboost is an iterative algorithm, a number of trials will be operated during which each time a new weak classifier will be selected. Weights are applied to the set of example during each iteration indicating its importance.

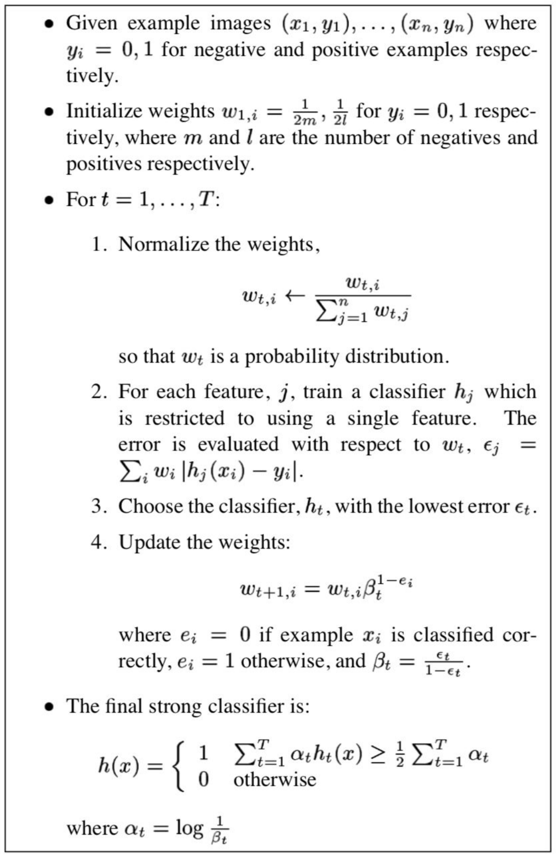
The main idea of AdaBoost learning algorithm is, choose the **most efficient weak classifier** every iteration (lowest error cost), wrong examples’ weights will be increased so that next weak classifier will “focus” more on the hard ones. Finally, the “strong” classifier is an combination of a small number of good classifiers.


f_t
θ_t
θ_t is a threshold for classifier h_t
$f_t(x)$
Outputs of a possible rectangle feature on training examples x (faces and non faces)

Resulting weak classifier:

$$h_t(x) = \begin{cases} +1 & \text{if } f_t(x) > \theta_t \\ \boxed{0} & \text{otherwise} \end{cases}$$

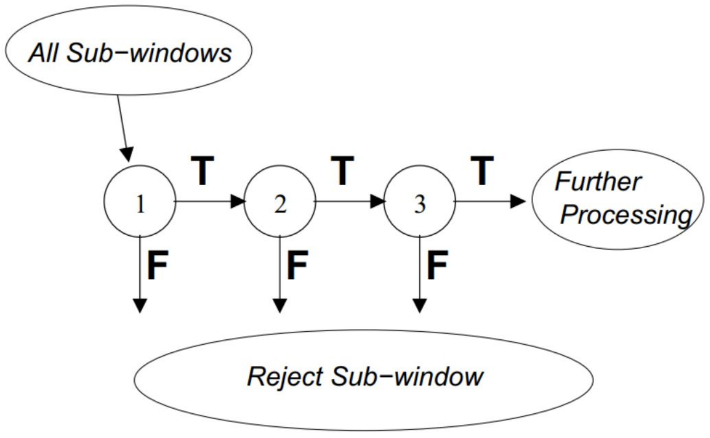
For next round, reweight the examples according to errors, choose another filter/threshold combo.



- Given example images $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i = 0, 1$ for negative and positive examples respectively.
- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0, 1$ respectively, where m and l are the number of negatives and positives respectively.
- For $t = 1, \ldots, T$:
  1. Normalize the weights,
     $$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$
     so that w_t is a probability distribution.
  2. For each feature, j, train a classifier h_j which is restricted to using a single feature. The error is evaluated with respect to w_t, $\epsilon_j = \sum_i w_i \left| h_j(x_i) - y_i \right|$.
  3. Choose the classifier, h_t, with the lowest error ϵ_t.
  4. Update the weights:
     $$w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}$$
     where $e_i = 0$ if example x_i is classified correctly, $e_i = 1$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
- The final strong classifier is:
  $$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$
  where $\alpha_t = \log \frac{1}{\beta_t}$

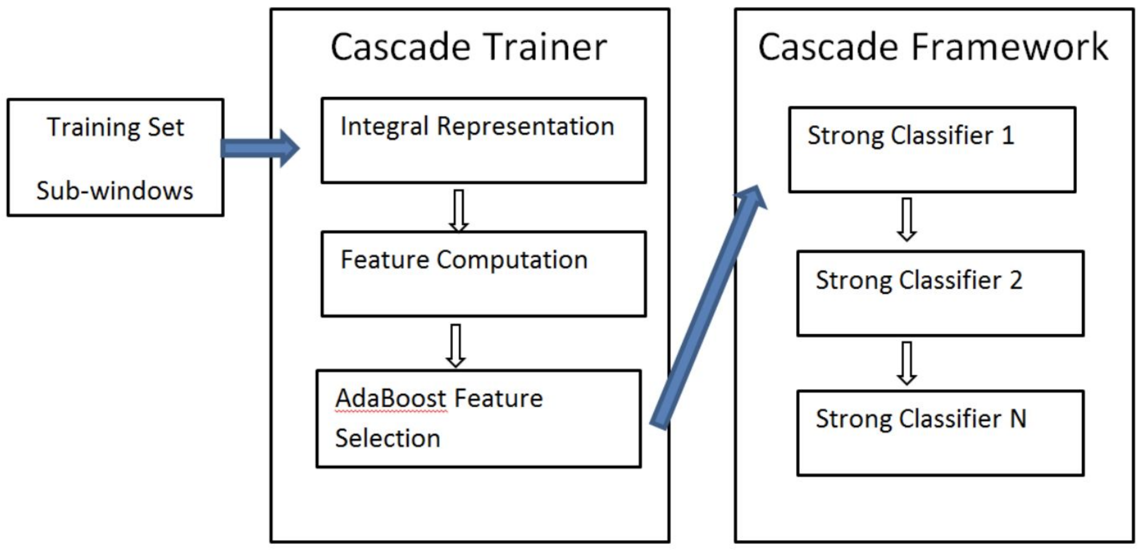
Table 1: The AdaBoost algorithm for classifier learning. Each round of boosting selects one feature from the 180,000 potential features.

The Attentional Cascade :



There are three main parameters of the cascade. The number of the layers (strong classifiers) in cascade, number of features in each strong classifier, threshold of each strong classifier. To get the optimized combinations of these three parameters is far more complex, cascade has to be designed gradually. Given P = positive example set, N = negative example set. = feature numbers for ith classifier.

Summary of Viola_Jones Algo :


Training Set
Sub-windows
Cascade Trainer
Integral Representation
Feature Computation
AdaBoost Feature Selection
Cascade Framework
Strong Classifier 1
Strong Classifier 2
Strong Classifier N